講　座

プログラム開発の工程管理 I*

大日方　真**

1. プログラム開発のポイント

(1) はじめに

プログラム開発の作業は，研究開発的な性格が強い．一般には，研究開発の成果がそのまま実用になることはまれで，実用化のためには，試作と改良の過程が必要である．改良を重ね，試作の段階を経て実用にたえうることが確かめられたうえで，はじめて商品化が行なわれる．アイデアを製品化するためには，試行錯誤を繰り返しながら，アイデアを取捨選択し，具体化していかなければならないはずである．

しかし，プログラムの場合には，他の製品とは異なって，研究開発的な作業の成果として生み出されたものが，それ自身試作品であり，かつ直ちに実用品として使用されることになる．研究開発の成果をすぐに実用化するということは，実用化の工程に多くの無理が含まれていることを意味している．実用化されたプログラムには，多くの不備と潜在的な故障が残るだろう．ときには，使ってみたら使いものにならなかったという，他の商品では考えられないような致命的な欠陥が含まれていることもある．プログラムが利用者の手に渡ってから次々にみつかるそのような欠陥のために，プログラムの開発にあたった人は，長期間にわたってその手直し（修理）に追いまわされる．

プログラムの工程管理をする際に，考えておかなければならないのは，プログラム開発工程に以上のような無理が含まれているという点である．しかし，無理があっても，開発工程を根本的に変えることはほとんど不可能である．それは次のような点を考えてみれば明らかである．

(i) 同一のプログラムを複数個，商品として提供できることは少ない．ほとんどの場合，1個だけの特製なので，プログラムの開発コストは非常に高くなる．したがって，ハードウェアの製品開発のような，念を入れた開発工程をとることは，開発コストの点からむずかしい．

(ii) 短期間にプログラムを完成させなければならない．情報処理部門の発展の速さが，とくに短期間の作業を要請している．

このようなプログラム開発工程の特質を十分に認識してはじめて，開発工程をどう改善したらよいかということが，切実な現実の問題となるのである．

(2) プログラム開発の現状

次に，プログラム開発工程の問題を整理し，改善の方向を模索してみよう．

プログラムの開発が研究開発的であり，担当者の開発に対する心情が，ときには芸術的ですらあるということは，プログラム開発にあたって，開発にたずさわる担当者の力量が，プログラムの良し悪しを左右する重要な要素になることを意味している．従来からプログラマの作業は，孤独な作業であった．独力でコツコツとプログラムの論理を組み立て，プログラムを記述する．デバグの作業では，プログラムに含まれている論理的な誤りや，プログラムを記述する段階で発生した不注意な誤りを除くために，極度の集中力と忍耐力を要求される．プログラムに含まれている誤りの数がわからないのであるから，デバグの作業がいつ終わるかを正確に予測することは不可能である．したがって，プログラムの完成目標日直前になっても，そのプログラムが実際にいつ完成するのかはっきりつかめない．プログラムの完成は，デバグの作業が長びくにつれてずるずると延びるのである．これまではプログラマ自身も，プログラムの依頼者やプログラマを取りまく周囲の人たちも，プログラムの作成とはそういうものだ

---

* The management of program development, by MAKOTO OBINATA (NIPPON SOFTWARE CO., LTD.)
** 日本ソフトウエア株式会社

と思い込んできた．そして第三者がプログラム開発の日程や品質を管理するなどということは，専門家としてのプログラマのプライドが許さないし，技術的にも不可能だと考えられてきたのである．

しかし，コンピュータが進歩し，情報処理の分野が拡大するにつれて，プログラマ個人の能力を信頼し，個人の力量にすべてを依存する方式には，いろいろな面で問題が出てきた．たとえば，オンライン・システムやデータ・ベースを扱うプログラミングでは，個人が独力ですべての作業をすることは不可能であり，グループによる共同作業が必要になった．また，プログラム開発を依頼する人の知識や経験が豊かになり，プログラムの完成期日や品質にきびしい条件がつけられるようになって，生産性の向上と品質の向上の問題に無関心でいることが許されなくなったのである．

指定された期日までにプログラムを完成させ，かつその品質まで保障しなければならないことになれば，どうしてもプログラム開発の管理の問題を考えないわけにはいかない．

歴史的に見ると，人間は物からエネルギーへ，エネルギーから情報の支配へと，活動の目標を転じてきた．プログラムは，情報処理，情報の支配という活動の要請として開発され，製品としての意味をもつようになった．プログラムの開発は，情報的生産と深いかかわりがあるのである．しかし，情報的生産という新しい生産を理解することは，必ずしも容易ではない．その理解の困難さがプログラム開発を必要以上に個人の技術力と良識にゆだねてしまうという自由放任主義に走らせていたのではないかと思われる．そこで情報的生産の特質を明らかにし，情報的生産においても，工業製品の生産の場で行なわれてきた生産管理の経験や手法を吸収し，その適用の仕方を考えることが重要なのではないだろうか．

情報処理の先進国であるアメリカでは，すでにプログラム開発管理についてのいくつかの手法が開発され，経験も蓄積されている．それはわが国にもいろいろな形で紹介されているが，プログラム開発のように本質的には人間に依存している問題は，その集団のもつ習慣や特性を無視しては，効率的な解決の仕方を考えることができない．独自の努力が必要なのである．

問題を明確にするために，次にプログラム開発の特徴を整理しておこう．

(a) プログラマ個人の技能に頼る

プログラム開発の作業は，プログラマ個人の技術力に頼る面が多く，作業が作業者の個性や個人差に影響されやすい．作業の進行状況を客観的に掌握しにくく，問題があっても適切な対策を講じにくい，作業の交代や共同作業もむずかしい．

(b) プログラマの不足

すぐれたプログラマは，すぐには育たないから，プログラム開発の要請に対して必要なプログラマの数を確保することがむずかしく，常にプログラマの不足に悩まされる．

(c) 生産性が低い

プログラム作成の作業は，計算機を使用するとはいっても，開発工程全体は，前近代的な家内手工業的生産の性格が強く，生産性を向上させることがむずかしい．

(d) 開発コストが高い

生産性が低ければ，開発コストは高くなる．プログラム開発のコストは，おもに人件費と計算機使用料である．予定以上に担当者を投入しなければならなかったり，作業期間が大幅にのびたりすると人件費がかさむことになる．また担当者の非効率的な作業は，直接計算機使用時間の増加としてはね返る．開発したプログラムが，仕事の手順や方法が変わって使えなくなったりすると，損失は大きい．しかし，一般にはそのような損失にあまり注意が払われていないようである．

(3) プログラム開発工程の分業化

ではこれからのプログラム開発は，どのような仕組みで行なったらよいのだろうか．

第1に考えるべきことは，近代工場における製品の生産形態にてらして，ソフトウェア開発工程を再検討することである．

歴史的に見ると，工業製品の生産工程は，生産の近代化が進むとともに細分化され，分業化と流れ作業による自動化が進められた．分業化が進むにつれて，組立作業に従事する作業者の作業は，単純化され，主として耐久力と注意力があれば，特殊な技能をもたなくても作業を担当できるようになった．このような工業製品の製造工程に準じて，ソフトウェア開発過程を細分化したらどうなるだろうか．分業化できれば，それぞれの工程の作業の質と作業量を高めるための管理が容易になる．また，担当者の技術力を限定できるので，比較的短時間にプログラム開発の担当者を養成できるだろう．

しかし，物的生産の仕組みがプログラム開発の研究開発的な性格と，はたしてどこまでなじむだろうか．

工場生産のような分業化を実現するためには，製品の設計図が詳細で，かつ正確でなければならない．プログラムの設計図は，機能仕様書や設計仕様書と呼ばれているが，これらの設計書はプログラム・テストの段階で，しばしば機能的に不備があることや，性能が低すぎることがわかって，修正が必要になるのである．ときには，プログラムが完成して広く使われるようになってから，機能的な欠陥，すなわち設計図の誤りが指摘されることもある．設計図がそのように不安定であるとすれば，開発工程をあまり細分化して分業化しても意味がない．それによって開発工程が合理化されるとはいえないだろう．

プログラムの開発にたずさわる人の中で，工場生産の場合の組立工にあたる人を，一般にコーダと呼んでいる．プログラム開発の分野では，システムズ・エンジニア（あるいはシステムズ・アナリスト），プログラマ，コーダ，オペレータという職種の分化が進みつつあるが，この場合コーダの専門性とはなんだろうか．プログラム流れ図を見ながら，正確に速くコーディングできれば，すぐれたコーダということになるだろう．しかし，流れ図（設計図）が不完全で，プログラム・テストの段階や実用の段階になってから，流れ図の修正が必要になり，その度にコーディングをし直さなければならないとしたら，コーディングの作業を合理化しても，生産性の向上に大した役割は果たさないし，コーダの意気もあがらないだろう．プログラムの設計図が不安定である限り，専門職としてコーダという職種を考えることは大変むずかしい．事実，これまでわが国では専門職としてのコーダが育つ余地はほとんどなかった．現在では，プログラマとコーダを一体として考えようとする考え方に，再びおちつきつつあるのが実情である．プログラム開発過程の分業化を進めるためには，その前提としてシステム設計やプログラム設計の質を高めなくてはならない．したがって，設計にたずさわるシステムズ・エンジニア，あるいはシステムズ・アナリストをとう養成するかということが課題となるのである．一般の工業製品の分野では，大学の理工系の学部で専門的な教育を受ければ，それで研究者や設計者としての素養を身につけたと考えられ，大学を卒業すると直ちに設計者としての道を歩むことになる．しかしプログラミングの分野では，大学の理工系で学んだことが，必ずしもシステムズ・エンジニアとしての資質や素養と結びつかない．システム設計能力は，情報処理についての実地の経験を積んではじ

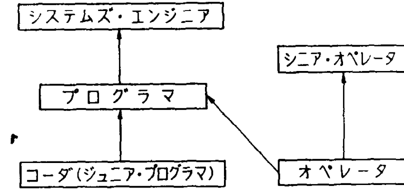

図 1

めて身につく部分が多く，実地の経験に乏しいものがいきなりシステムズ・エンジニアとしての力を発揮することは不可能である．実地の経験とプログラムの設計能力という点から考えると，オペレータ，コーダ，プログラマ，システムズ・エンジニアという各職種を階層構造にし，システムズ・エンジニアを頂点として，たてのパイプで結んだ開放的な組織にすることが望ましい（図1参照）．

コーダは，努力によってやがてプログラマになれ，プログラマは，システムについての経験を積むことによってシステムズ・エンジニアになれる．そして，プログラム開発にかかわる責任の範囲を次第にひろげていくことができるということが，コーダのコーディング意欲や，プログラマのプログラム開発に対する取組みをもりたてることになるだろう．

システムズ・エンジニアになれる人は，ごく限られた人たちであると思われがちであるが，大半のプログラマは，機会さえ与えられれば，経験を積み重ねてシステム設計を担当できる能力を身につけ，標準的には5年程度でシステムズ・エンジニアとして一人立ちできるようになるだろう．

プログラマやコーダに対して，設計にたずさわれる道が常に開かれており，またそれが期待されているとすれば，それは工場生産の分業形態との決定的な違いであり，そのような分業形態が，プログラマ1人1人のプログラム開発能力を高めることになる．そういう点をよくふまえたうえで，プログラム開発の管理のあり方を考え直してみなくてはならないのではないだろうか．

（4） 教育的な配慮

プログラム開発の作業が研究開発的な性格が強いということは，当然のことながら，開発の仕組みを改善しようとするとき，工場生産の形態を模倣するだけでなく，他方では，研究開発体制を強化し，研究開発にふさわしい条件を準備することが合わせて必要になるのである．計算機導入の比較的初期の時代であった，

1960 年代の前半は，プログラマは適用業務別に担当を割り当てられ，その業務については関連部局との折衝からシステム設計，プログラミング，しばしば機械化実施後のオペレーションまで，すべての責任をもたせられるのが普通であった．そのような経験をする機会にめぐまれたプログラマは，計算機についての幅広い技術を身につけ，システムズ・エンジニアやシニア・プログラマとして成長し，各企業において貴重な人材となっている．

しかし，機械化が進むにつれて，プログラムの規模が大きくなり，複雑度も増し，同一のプログラマがプログラム開発の全工程を担当することは不可能になり，1人1人のプログラマの作業の範囲や，果たすべき役割は必然的に限定される．その結果，研究開発にたずさわる人に期待される幅広い能力が育ちにくいことになったのである．

たとえば，詳細設計図を与えられて，それを見ながらプログラムをコーディングしている人たちは，いくらコーディングの経験を積み重ねても，社会や企業に対して計算機がどう貢献しているかという問いかけに答えられるようにはならないだろう．また計算機の新しい適用業務を開発する力をもつこともできないだろう．すなわち，プログラム開発作業の分業化とともに生れた専門的な職種の間を，風とおしのよいパイプでつなぎ，機構的には職種間の流動性を保障したとしても，1つの職種での実地経験の積み重ねが，必ずしも他の職種で要求される能力を高めることに結びつかないということになるのである．そこで，コンピュータ要員の能力をひろげていくためには，プログラム開発の全工程に責任と権限をもつ人による特別な配慮が必要なのである．たとえば，現在もっている個人の能力やスキルを正しく掌握し，それを十分に活用できるような仕事の割り当てをしなくてはならない．同時に，個人の能力を高めるための教育的配慮が必要である．適正配置や適切な教育を行なうためには，スキルズ・インベントリが重要である．スキルズ・インベントリでは，個人の技能や特技，個人の仕事に対する希望などをできるだけ正確に申告させて管理する．個人の作業実績を掌握するためには，作業管理や進度管理のデータが必要である．作業完了後の評価や教育訓練を受けたときの成績も貴重なデータになる．

詳細流れ図を見てプログラムのコーディングをする作業に対して自信をつけた人には，やがてあらい流れ図からプログラミングができるようになることを期待して，新しい仕事を割り当てる．プログラミングについて十分な力をもった人には，機能仕様書の作成などの作業に参加させて，設計にたずさわる能力を身につけることを期待する．そのようなプログラマの成長に期待をかけた管理方法や開発の仕組みを考えるのである．プログラマの苦手な文書化の能力なども，教育的な配慮を重ねることによって高めることができる．

プログラマが自分の作業量を正しく見積り，その時々の作業状況を正しく報告できるようになるためには経験が必要である．そのような力は，各人に毎週あるいは毎月，作業状況の報告を提出させて，その内容を分析し，前回の報告内容や，他の人の報告との関連を調べ，矛盾があれば本人に差し戻し，再検討させるという操作を繰り返すことによって高めることができる．

プログラミングの全工程は，教育の過程なのである．教育的配慮を欠いて，プログラミングの工程管理やプログラマの管理を行なっても，はたして実効があがるかどうか疑わしい．どうしたら個人の能力を十分に発揮させることができるか，どうしたら個人の能力を高めることができるか，という自らに対する問いかけが，プログラム開発の工程管理にあたるものにとっては最も重要なのである．

2. プログラムの工程管理

プログラム開発の工程を大ざっぱに分けると

I．プログラムの設計

II．プログラムの作成

という2つの段階を考えることができる．

プログラム設計の段階には，開発の対象がシステムのときは，システムの分析やシステム設計の作業も含まれる．プログラム設計作業のゴールは，プログラムの設計仕様書を作成することである．プログラム作成の段階では，プログラム流れ図の作成，コーディング，デバギング，文書化などがおもな作業になる．

そこで，次にこれらの各段階の作業について，技術的な問題を少し考えておくことにしよう．技術上の問題は，各工程の作業の進度と密接な関係があるからである．

(1) プログラムの設計

プログラムの設計概念として，最近，モジュール構造とかブロック構造とか，開放的な (open-ended) 構造という用語がよく使われるようになった．プログラムに要請される機能が複雑になるにつれ，1つのプロ

グラムが，ときには数千ステートメント，数万ステートメントにも及ぶ場合が出てきた．プログラムの規模が大きくなるにつれて，当然プログラムを作成したり，プログラムを修正したりする負担は加重される．そこで，複雑なプログラムについては，必要とされる機能をいくつかの機能要素に分解して，それぞれの機能要素を積み重ねることによって，全体としての機能をもたせるという方式を考えた．そして，それぞれの機能要素ごとに，その機能要素を折り込んだサブプログラムを作る．そのようにして設計された全体のプログラムの構成要素であるサブプログラムを，モジュールと呼ぶのである．

通常，プログラムは全体の制御を行なう主プログラム1つと，特定の機能要素に対応して特定の処理を行なうサブプログラムが複数個集って構成される．

1モジュールの大きさは，理想の状態としては，記号言語（アセンブリ言語）による場合には，100 から 200 ステップ程度，問題向き言語の場合には，コメントを含めてソース・リスト 20 ページ以内におさえるようにするのがよいとされている．

各モジュールは，標準のリンケジによって結びつけられ，モジュール間で情報のやりとりが必要なときは，パラメータを介して受け渡しをする．

プログラムをモジュール構造にして組み立てると，各モジュールごとに独立して開発を進めることができるから，プログラムの作成，テスト，保守が容易になる．したがって，プログラム開発の作業工程を短縮できるのである．

しかし，プログラムをモジュール構造にするためには，いくつかの前提条件が必要である．たとえば，プログラム設計を担当できるすぐれたプログラマが必要である．各モジュールに盛り込む機能要素や，各モジュール間の関係を，プログラミングに先立って完全に決めておかなければならないからである．

また，文書化を重視する作業方式がとられていたり，作業の進め方や文書化の仕方についての標準化が進んでいたりしなくてはならない．担当者各人が，思い思いの方式をとることが認められていたり，プログラムのデバグが終わってプログラムが完成してから，暇をみて文書化を行なう習慣が残っていたりするところでは，プログラムをモジュール構造にすることは不可能である．

プログラム設計の段階でもう1つ重要なことは，最適化についての配慮である．最適化は，ファイルの設計，データ・ハンドリング，データの加工や編集など，あらゆる場面で問題になる．最適化に対する考慮を欠いたプログラムは，利用者の満足を得られないことが多く，利用者の手に渡ってから根本的な改訂を要求されることになる．

（2） コーディング

プロセス制御のプログラムや，オンライン・システムの制御プログラムの場合のように，数マイクロ秒の遅れでも処理に問題が出る場合は別として，一般的には，いたずらにコーディングの技術だけを競うような記述は歓迎されない．プログラムはできるだけわかりやすい記述にすべきである．そのためには，直線的なコーディングがよい．プログラムをわかりやすくすればするだけ，プログラムの修正や拡張が容易になる．命令の一部を定数として使ったり，相対アドレスを乱用したり，命令部を実行時にむやみに修正したりすることは避けるべきである．

命令の名前やデータの名前のつけ方によっても，プログラムをわかりやすくすることができる．処理の内容と関連ある名前をつければ，名前を見て処理を判読できるようになる．プログラムの各ステップを，プログラム流れ図のステップと対応させ，プログラムの中にはコメント（注釈）やリマーク（備考）を，できるだけ多く挿入する．プログラムを読む立場で考えると，問題向き言語では，平均してステートメント1行に対して1行のコメントやリマークを書く．記号言語で記述したプログラムは，各命令の余白に必ずリマークを書き，かつ適当なブロックごとにそのブロックに関する説明を，コメントとして挿入しておけば理想的である．

問題向き言語を使う場合，それが機械語に翻訳される仕組みを知っていれば，効率的な目的プログラムが生ずるようなコーディングを考えることができる．このような細かい配慮は，使用頻度の高いルーチンを作るときに必要である．

コーディングの際に重要なことは，プログラムのデバギングができるだけ容易になるように考えてプログラムを記述することであり，そのような配慮によってプログラムの完成を早めることができる．たとえば，作業区域や定数の保存域を1箇所にまとめたり，診断用のルーチンをあらかじめ組み込んでおくなどの配慮はその1例である．

（3） 文書化

プログラムの開発に伴って，多くのドキュメント（文

書）が作られる．

プログラム開発の生産性を向上させるための有力な手がかりとして，最近文書化が重視されるようになった．ドキュメントが不完全であると，情報の交換や伝達が十分に行なえない．そのために誤解に基づく製作があとをたたず，プログラムの作り直しや，手直しが必要になって，それが生産性を低めているからである．プログラミングの過程で作られるドキュメントには，次のようなものがある．

（i） プログラム開発過程で作られるドキュメント．——設計仕様書や技術的な打合せのために用意する資料など．

（ii） 開発されたプログラムの利用者のために作られるドキュメント．——プログラムの解説書や操作説明書など．

（iii） 保守のためのドキュメント．——プログラム完成後の手直しや改訂に備えて整備しておくドキュメント．原始プログラムを入力すると，流れ図を作成して出力してくれる流れ図作成プログラムは，保守用のドキュメントを作るのに有効である．

（iv） 教育用のドキュメント．——プログラムの内容，価値，使い方などを説明した教材など．

（v） 価値を伝達するためのドキュメント．——PR用の資料や論文など．

（vi） 経験を蓄積するためのドキュメント．——プログラム開発に要した日数，人員，作業時間，コンピュータ使用時間，作業中に生じた問題点などを記述する．

以上のようなドキュメントは，他の技術文書に比べていくつかの特殊性をもっている．たとえば，次のような点である．

（i） 文章を書くことについて関心の薄い人がドキュメントを書く場合が多い．

（ii） プログラム開発の作業は，時間的にも金銭的にも余裕のないことが多く，そのために担当者が文書化の作業量を極端に切りつめてやりくりをするということに走りやすい．

（iii） 用語に外国語や説明を必要とする新語が多く，日本語としての適切な表現のない用語や，意味や表記法の不統一な用語を使わざるを得ない．

（iv） ドキュメントの標準的な編集方式や表記法が確立されていない．

（v） しばしば改訂が必要である．

これらの特性を考えたうえで，プログラムのドキュメントの作り方の原則を確立しなくてはならない．ただいたずらに，プログラミングの作業の中での文書化の重要性を強調しても，作業の負担だけが増加し，実効はあがらないということになるだろう．

文書化について留意すべき点をいくつかあげておこう．

（i） 必要最低限度の標準仕様を作る．

あまり細かい規定を作り，規則でしばりすぎると作業効率が低下する．

（ii） 文書化の作業量について適切な見積りをする．

一般に，プログラム開発作業の中で占める文書化の作業量が少なく見積られすぎている．プログラム開発の全作業中に占める文書化の作業量は，しばしば20%をこえると考えるべきである．

（iii） ドキュメントの検査体制を確立する．

標準仕様のとおりかどうかを検査し，検査結果をドキュメント作成者に知らせて改善させたり，必要ならば標準仕様や検査の仕方を改める．

（iv） ドキュメントの登録・保管・改訂などの手続きを取り決めて実行する．

（v） 文書化についての教育を行ない，標準仕様を身につけさせ，文書作成能力を高める．

（vi） 一度ドキュメント中心のプログラム開発方式をとったら，その方式に徹する．ドキュメントが作られたら，そのドキュメントを維持するために，どんな細かい点の変更に対しても，直ちに情報の追加や内容の更新を行なって，ドキュメントを書き改めなくてはならない．デバギングの作業中に発見された，論理的な誤りを修正するために，原始プログラムを差しかえたとき，流れ図をそのままにしておいたため，それまで丹精をこめて記述した流れ図が，それ以降役に立たなくなるということがよく起こる．

（4） デバギング

プログラム開発工程の中で，デバギング作業の工程管理はとくに重要な意味をもっている．ものによってはデバギングの作業量が，プログラミングの全工程の60%に及ぶことがあり，デバギングの作業は大きな比重を占めているからである．したがって，デバギングの工程を短縮することが，プログラミングの工程全体を短縮することにつながり，それがプログラム開発管理の主たるねらいだといっても過言ではないだろう．

プログラムの誤りは，プログラム作成者自身の手で

見つけ出さなくてはならないが，最近では担当者のデバギング作業の負担を少しでも軽くするために，デバギングの道具になるソフトウェアが開発されるようになり，プログラムのエラー検出の作業も自動化の度合を深めつつある．

プログラムのデバグを効率的に行なうためには，次のようなことを考えておくと有効である．

(i) プログラムをモジュール化し，各コンポーネントはできるだけ単純な，わかりやすい構造にする．

(ii) プログラム・テストを部分テスト，つなぎテスト，総合テストの順に段階的に行なう．

(iii) 机上テスト (desk debug) を十分行なう．

(iv) テスト・データを単純なものから複雑なものへと，順を追って用意し，プログラムの機能を1つずつ確実にテストできるように配慮する．

(v) テスト・データとして，誤りのデータや例外事例なども用意する．

(vi) プログラム・テストの道具をうまく使う．プログラム・テストのためのサポート・プログラムがいろいろ開発されている．

プログラムの開発にあたっては，問題の解析，流れ図の作成，コーディングなど作業のすべての段階で，誤りをおかすおそれがある．しかし，問題の大まかな分析から細かい分析へ，さらにコーディングへと，細心の注意を払って進めば誤りを最小限度にとどめることができるだろう．誤りをプログラムの中にまぎれこませないためには，細心の注意を払うとともに，一段階の作業を終わったあと，その内容を人手によって十分チェックしてから先に進むことが必要である．人手による机上チェックには，次のようなチェックが含まれる．

(i) 問題が正しく記述されているかどうかを調べる．

(ii) 問題を解くために最適な解法を使っているかどうかを調べる．

(iii) 流れ図に示された論理が正しいかどうかを調べる．とくに例外的なデータや処理に気をつける．コントロールの合流点や判断を行なう場所に注意する．

(iv) ループを調べる．ループの中には初期値設定，ループの中で行なう演算処理，ループを終結するための判定，ループに飛び込む，またはループから飛び出すためのブランチ命令などが必要である．ループを何回まわるかを調べて，その回数が正しいかどうかを見る．

(v) 流れ図を2度書いてみる．流れ図の当否を調べるのによい方法は，1～2日後にもう1度問題の記述に従って流れ図を書いてみることである．この方法はコーディングのチェックにも有効である．

(vi) 問題分析段階の誤りに注意する．問題分析の段階で生じる誤りは，普通論理的な誤りである．この種の誤りは比較的少ないが，プログラム作成の初期の段階で起こるので，見逃しやすく，方法を誤解したための誤りだから見つけにくい．

(vii) 流れ図とコーディングを対照して調べる．流れ図とコーディングとを対照させながら調べる方法は，誤りを発見するための最もよい方法の1つである．

(viii) 別な人がチェックする．コーディングから逆に流れ図を書き，それがもとの流れ図と一致しているかどうかを見ることによって，いろいろな誤りを見つけ出すことができる．問題分析の結果を口頭で説明したり，別な人に同じプログラムを書かせたりすれば，ほとんど確実に論理的な誤りを除くことができるし，場合によってはコーディングの誤りまでわかる．

(ix) コントロール・カードの間違いなど物理的な誤りを調べる．

(x) 最新のプログラムかどうかを調べる．

机上デバグを無視し，安易にコンピュータに頼るデバグは一般に非効率的である．机上デバグでは，1度に複数個の誤りを発見することはさして困難ではないが，コンピュータに依存したデバグでは，すぐれたテスト・ファシリティが用意されている特別なコンピュータを使用するのでない限り，1度に1箇所の誤りしか発見できない．その点からすれば，机上デバグにできるだけ比重をかけることが望ましいのである．

一般的におかしやすい誤りは，論理的な誤り，コーディングの誤り，物理的な誤りのいずれかということになるが，それらのほとんどが机上デバグで発見できるだろう．

論理的な誤りには，たとえば次のようなものがある．

(i) 判断の結果の行先をとり違える．

(ii) ループを終結させるテストが不適当である．

(iii) ループの回数に間違いがある．

(iv) ループの開始手続きを抜かしたり，間違えた

ぴする.

（v） 計算中に起りうるすべての場合や，データとしてはいってくる可能性のあるすべてのものを考えていない.

コーディングの誤りには，部分的なコーディングのし忘れ，定義していない記号や重複して定義した記号の参照，記号や命令コードの綴りの間違い，入出力の指定やサブルーチンの呼出し方の誤りなど，初歩的な誤りが多い．これらの大半は，アセンブルやコンパイルをしてみれば直ちに見つかるもので，アセンブルやコンパイルを机上デバグと併用すると有効である.

物理的な誤りには

（i） データ・カードを忘れた.

（ii） カード上のデータの形式を間違えた.

（iii） プログラム・デックの相対的な順序を間違えた.

（iv） プログラム・デック内のカードの位置を誤った.

（v） コントロール・カードの入れ忘れや，打ち間違い.

（vi） アセンブルし直したデックを使ったとき，もとのデックに対する変更カードをすてるのを忘れた.

など，不注意による誤りが多い.

（5） プログラムの保守

プログラムが完成したあとも，処理手順の不備が発見されたり，処理方法が変更になったり，プログラムの適用範囲が拡大されたりして，プログラムの手直しが必要になることが多い．このようなプログラムを手直しする作業を，プログラムの保守といっている．プログラムが複雑になればなるほど，保守の負担は大きくなる．プログラム開発の全コストのうち 50% 以上が保守に費やされることもめずらしくない．プログラム開発の全工程がプログラマ個人の手にゆだねられていたときには，プログラム作成者がそのプログラムの保守をも担当していた.

しかし，プログラム作成者がプログラム完成後も，保守担当者として拘束されることになると，他のプログラム開発の作業を担当させるときに支障をきたす．また，引き続いて保守を担当することがわかっていると，どうしても文書化の作業を怠りがちになる．そこで，プログラムの開発が進むにつれて，プログラムの保守を開発と切り離して管理し，専門の保守担当者の手でプログラムの改訂を行なうことが必要であると考えられるようになった．改訂の必要性が生じたら，ごくわずかの変更でも必ず理由を書いた文書で申請し，責任者の承認を得たうえで保守担当者に改訂を依頼する．修正が行なわれたら，その内容は関係者やそれによって影響を受ける人たちに直ちに知らされる．とくに，プログラムの論理構造や処理方法を変えるときには，慎重な配慮が必要である.

プログラムの修正は，まず流れ図の修正からはじめ，改訂された流れ図に従って原始プログラムを修正し，流れ図やプログラム・リストは，常に最新の状態にしておく.

（6） プログラムの検査

プログラムの質を高めるためには，完成したプログラムが一定の水準に達しているかどうかを，第三者の手で確認するプログラム検査の制度が必要である．プログラム検査を組織的に行なえば，単にプログラムの質の保障だけでなく，担当者の技術的に未熟な点が指摘でき，プログラマの技術水準の向上をはかることにも貢献できる．プログラム検査にとって重要なことは，プログラムは，必ず第三者によって見直され，チェックされなければならないという考え方を徹底させることである．この意味で重要なのは，作業グループ内の検査体制を確立することである．たとえば，10 人からなる作業グループの組織は**図 2** のように考える.

S_0 はグループ・リーダである．グループ・リーダはグループ内の指示系統を明確にするために，必要に応じてグループ内の中間管理者を定め，(S_1, S_2, S_3 など）ワンマン，ワンボスの体制を作る．この図 2 の例では，S_0 は S_1, S_2, S_3 に対して各モジュールの設計指針を示して，モジュールの設計を指示する．モジュールを構成するコンポーネント（ルーチン）の機能仕様と設計の概要は，S_1〜S_3 が作り，各コンポーネン

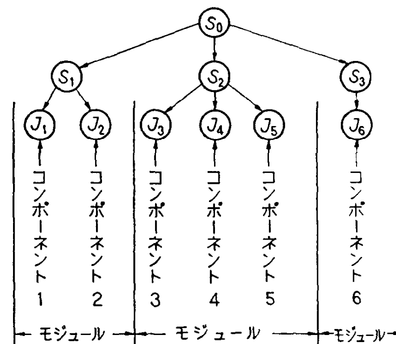

図 2　作業グループ内の組織

トの担当者にそのプログラミングを指示する．したがって，S_0 は各モジュールの検査について，S_1, S_2, S_3 は各コンポーネントの検査についての責任をもつことになる．そしてプログラム全体は，プロジェクト・リーダの上司とプログラム作成依頼者による総合検査を受け，その検査を通ってはじめて完成ということになるのである．以上のように，グループ内での個人の責任と権限を明確にすることが，プログラム検査の出発点である．

次にモジュール検査，コンポーネント検査，総合検査の検査規準の例を示しておこう．

(a) 各モジュールの検査規準

- モジュールの設計指針と矛盾していない．
- 要求された機能をすべてもたせている．
- 次の点について問題がない．
  - 効率
  - 拡張性
  - 記憶容量
- 用語が正しい．
- 仕様の決定している部分と，未決定の部分を明確に区別している．

(b) コンポーネントの検査規準

- 必要十分なテスト・データでテストした．
- テスト・データに矛盾がない．
- コンポーネント検査仕様書を作っている．
- 実行時間の測定（必要な場合のみ）を適当なデータで行なっている．

(c) 総合テストの検査規準

- プログラムの設計指針と矛盾していない．
- すべての機能をテストした．
- 誤りのデータ，機械の誤動作，操作誤りに対して適切な処置がとられている．
- 満足すべき効率を備えている．
- すべてのコンポーネント・テスト，モジュール・テストに合格している．
- ドキュメントが整っている．

(7) プログラミング技法の開発

現在のソフトウェア開発の方向として，ユーザのプログラミング負担を少しでも軽くすることが，1つのねらいになっている．プログラミングの負担を軽くするために記号言語が生まれ，FORTRAN, COBOL, ALGOL, PL/I などの問題向き言語が生まれた．また，レポート・ジェネレータやファイル・ジェネレータの開発が進んだ．ジェネレータのねらいは，作業条件を計算機に与えて，その条件に合ったプログラムを計算機に作らせることにある．作業条件を与えると，COBOL 言語で記述されたプログラムを出力してくれる Computer Science Corp のソフトウェア・パッケージ，COGENT II や COGENT III は，ジェネレータの考え方を発展させたものだということができる．こうした新しい技術の開発によってプログラミングの負担はますます軽くなるだろう．

ハードウェアの進歩につれて，計算機の記憶容量が増加し，処理速度ははやくなり，他方では計算機の価格が下がった．したがって，利用者は年とともに，より大型で，より高性能の計算機を利用することができるようになったのである．そうなると，多少プログラムにむだがあっても，そのむだがプログラムの実行に与える影響は少なくなるから，プログラミングの技法にあまり気を使っても意味がない．プログラムに多少むだがあっても，それはそのままにしておいて，プログラムを少しでもはやく完成し，使用したほうがよいということになる．プログラムの性能を高めるために要する労力や時間のほうが問題になりだしたのである．

ところで，プログラミング技術の開発は，創造力に富んだすぐれたプログラマの開発努力に依存している．したがって，プログラミング技術の開発のためにもっとも重要なことは，すぐれたプログラマを養成する教育であるということができる．経験を活用したり，個々の技術を有機的に結合させるためには，専門の開発グループを作ったり，セミナーや自由討論の場を作ることも必要である．

プログラミングの開発にたずさわる技術者は，これからは技術者としての技術的な能力を持っているだけではつとまらなくなるだろう．狭い意味でのプログラミング技術だけでなく，社会科学の常識や社会工学的な物の見方，さまざまな業務の実務経験などが必要となるのである．新しい技術を開拓できる技術者は，計算機部門の専門家という狭い観念から脱却して，新しい知的技術者に転化を迫られているといえる．しかし，専門技術についての知識とともに，ひろく関連諸科学に関する基礎知識と，豊かな教養を合わせて身につけた技術者，しかも日々新たに開発されていく新しい技術を，独力で習得できるような研究的な態度や，意欲を身につけた技術者の養成は容易ではない．

(8) プログラム開発の工程管理

プログラム開発の工程管理を考える場合，以上に述

べてきたように，各工程の作業の質を高めることが必要であり，その意味で技術的な問題が重要である．技術的な水準の高まりがないまま，単に管理の問題として工程管理を考えても，工程を短縮することはむずかしい．管理の問題としては適切な日程計画を立て，作業開始後は，進行状況を正確に掌握し，計画どおりに開発作業が進むように手立てを考えることが必要である．しかし，どんなに精密を計画を立てても，作業は計画したとおりにはなかなか進まない．作業をとりまく状況が絶えず変化するからである．その場合，状況の変化を客観的にとらえてくれるのがネットワーク技法である．一般に，ネットワーク技法は PERT で作成されるので，PERT の基本的な技法を使うと有効である．計画は，アロー・ダイヤグラムとバーチャートを使って記述する．

日程計画を立てる方法として一般的に行なわれている方法には，2つの方法がある．1つの方法は，作業の担当者にそれぞれの仕事に必要な時間を報告させ，それをもとに全体の計画を立てるのである．第2の方法は，完成目標日から逆算して，日程計画を求める．そのようにして作られた日程がもし個々の作業者にきついと感ずるものであれば，実施は困難なので，そのときは人員を増加したり，プログラムの仕様を縮小したりして，その日程を消化できる条件を作り出す．

いずれの場合にも，作業時間を見積るときには，それぞれの担当者や工程について最可能値を予測して，そのデータを使うことになるが，その他に，楽観値・最可能値・悲観値の3つの値を予測して，その3つの値を一定の式に代入して，期待値を求める方法もある．

工程管理を行なうときには，定められた様式に従って，作業者各人から作業状況の報告を求めることが必要である．それをもとに計画の進行状況をつかみ，必要な措置をして，次の作業ステップを管理するための資料を更新する．PERT のアロー・ダイヤグラムを使って工程管理を行なうとき，重要なのは，クリティカル・パス（最長経路）の管理である．クリティカル・パスは余裕日数が0の工程であり，このパス上の作業の遅れは，他の全作業の遅れを引き起こすことになるからである．工程管理の場合に重要なことは，いかにして作業進行の状況を定量化して掌握するかということである．プログラムの開発工程は定量化がむずかしいため，完成予定直前まで，作業の遅れがわからないということが起こりがちなのである．

次号では，プログラム開発の見積りの問題と作業管理について述べる予定である．（続く）

（昭和46年5月4日受付）